

## Fast

# Text Searching

## Allowing Errors

The string-matching problem is a very common problem. We are searching for a string  $P = p_1p_2\dots p_m$  inside a large text file  $T = t_1t_2\dots t_n$ , both sequences of *characters* from a finite character set  $\Sigma$ . The characters may be English characters in a text file, DNA base pairs, lines of source code, angles between edges in polygons, machines or machine parts in a production schedule, music notes and tempo in a musical score, and so forth. We want to find all occurrences of  $P$  in  $T$ ; namely, we are searching for the set of starting positions  $F = \{i | 1 \leq i \leq n - m + 1 \text{ such that } t_it_{i+1}\dots t_{i+m-1} = P\}$ . The two most famous algorithms for this problem are the Boyer-Moore algorithm [3] and the Knuth Morris Pratt algorithm [10]. There are many extensions to this problem; for example, we may be looking for a set of patterns, a pattern with “wild cards,” or a regular expression. String-matching tools are included in every reasonable text editor, word processor, and many other applications.

In some instances, however, the pattern and/or the text are not exact. For example, we may not remember the exact spelling of a name we are searching, the name may be misspelled in the text, the text may correspond to a sequence of numbers with a certain property and we do not have an exact pattern, the text may be a sequence of DNA molecules and we are looking

for approximate patterns. The approximate string-matching problem is to find all substrings in  $T$  that are *close* to  $P$  under some measure of closeness. The most common measure of closeness is known as the edit distance (also the Levenshtein measure [13]). A string  $P$  is said to be of distance  $k$  to a string  $Q$  if we can transform  $P$  to be equal to  $Q$  with a sequence of  $k$  in-

sertions of single characters in (arbitrary places in)  $P$ , deletions of single characters in  $P$ , or substitutions of characters. In some cases we may want to define closeness differently. For example, a policeman may be searching for a license plate ABC123 with the knowledge that the letters are correct, but there may be an error with the numbers. In this case, a string is of distance 1 to ABC123 if only one error occurs and it is within the digits area. Maybe there are always three digits in a license plate, in which case only substitutions are allowed. Sometimes one wants to vary the cost of the different edit operations, say deletions cost 3, insertions 2, and substitutions 1.

Many different approximate string-matching algorithms have been suggested ([4, 5, 6, 8, 11, 12, 16, 19, 20] is a partial list). In this article we present a new algorithm which is very fast in practice, reasonably simple to implement, and supports a large number of variations of the approximate string-matching problem. The algorithm is based on a numeric scheme for exact string matching developed by

Baeza-Yates and Gonnet [2]. The algorithm can handle most of the common types of queries, including arbitrary regular expressions, and several variations of closeness measures. In our experiments, the algorithm was at least twice as fast as other algorithms we tested (which are not as flexible), and for many cases an order of magnitude faster. For example, finding all occurrences of *Homogenos* allowing two errors in a 1MB bibliographic text takes about 0.4 sec on a SUN SparcStation II. We actually used this example and found a misspelling in our text.

The algorithms in this article served as a basis for a software package for Unix called *agrep*, which has been in use since June 1991. *Agrep*, which at the date of this writing is at version 2.04, is available by anonymous ftp from cs.arizona.edu. We describe *agrep* briefly at the end of this article. For more information see [15, 23, 24].

We begin by describing the algorithm for the pure string-matching problem (i.e., the pattern is a simple string). We then present many variations and extensions of the basic algorithm, culminating with matching arbitrary regular expressions with errors. Experimental results are given, as are details of *agrep*.

### The Algorithm

We first describe the case of exact string matching. The algorithm for this case is essentially identical to that of [2]. We present it here for completeness and because we use a slightly different notation. We then show how to extend the algorithm to search with errors. We then describe how to speed up the search with errors by using a *filtering search technique*.

#### Exact Matching

Let  $R$  be a bit array of size  $m$  (the size of the pattern). We denote by  $R_j$  the value of the array  $R$  after the  $j$  character of the text has been processed. The array  $R_j$  contains information about all matches of prefixes of  $P$  that end at  $j$ . More

precisely,  $R_j[i] = 1$  if the first  $i$  characters of the pattern match exactly the last  $i$  characters up to  $j$  in the text (i.e.,  $p_1 p_2 \dots p_i = t_{j-i+1} t_{j-i+2} \dots t_j$ ). When we read  $t_{j+1}$  we need to determine whether  $t_{j+1}$  can extend any of the partial matches so far. For each  $i$  such that  $R_j[i] = 1$  we need to check whether  $t_{j+1}$  is equal to  $p_{i+1}$ . If  $R_j[i] = 0$  then there is no match up to  $i$  and there cannot be a match up to  $i + 1$ . If  $t_{j+1} = p_1$  then  $R_{j+1}[1] = 1$ . If  $R_{j+1}[m] = 1$  then we have a complete match, starting at  $j - m + 2$ , and we output it. The transition from  $R_j$  to  $R_{j+1}$  can be summarized as follows:

Initially,  $R_0[k] = 0$  for all  $k$ ,  
 $1 \leq k \leq m$ ,  
 and  $R_j[0] = 1$ ,  
 for all  $j$ ,  $0 \leq j \leq n$ .

$$R_{j+1}[i] = \begin{cases} 1 & \text{if } R_j[i-1] = 1 \\ & \text{and } p_i = t_{j+1} \\ 0 & \text{otherwise} \end{cases}$$

If  $R_{j+1}[m] = 1$  then we output a match at  $j - m + 2$ ;

This transition, which we have to compute once for every text character, seems quite complicated. Other fast string-matching algorithms avoid the need to maintain the whole array. But, as observed by Baeza-Yates and Gonnet, this transition can be computed very fast in practice as follows. Let the alphabet be  $\Sigma = s_1 s_2, \dots, s_{|\Sigma|}$ . For each character  $s_i$  in the alphabet we construct a bit array  $S_i$  of size  $m$  such that  $S_i[r] = 1$  if  $p_r = s_i$ . (It is sufficient to construct the  $S$  arrays only for the characters that appear in the pattern.) In other words,  $S_i$  denotes the indexes in the pattern that contain  $s_i$ . It is easy to verify now that the transition from  $R_j$  to  $R_{j+1}$  amounts to no more than a *right shift* of  $R_j$  and an AND operation with  $S_i$ , where  $s_i = t_{j+1}$ . So, each transition can be executed with only two simple arithmetic operations, a shift and an AND. We assume that the right shift fills the first position with a 1. If only 0-filled shifts are available (as is the case with C), then we can add one more OR operation with a mask that has one bit. (Baeza-Yates and Gonnet used 0 to indicate a match and an OR opera-

tion instead of an AND; that way, 0-filled shifts are sufficient. This is counterintuitive to explain, so we opted for the easier definition.) An example is given in Table 1a, where the pattern is *aabac* and the text is *aabaacaabacab*. The masks for *ab* and *c* are given in Table 1b.

This discussion assumes, of course, that the pattern's size is no more than the word size, which is often the case. If the pattern's size is twice the word size, then four arithmetic operations will suffice. If the pattern is very large (e.g., in some biological applications the pattern size can be in the thousands), then other methods will be better. For most text-searching problems, small patterns are the norm. We discuss this issue later, but for now we will assume that the pattern's size is no more than the word size. This algorithm is clearly very easy to implement. Its running time is totally predictable because it essentially depends only on the size of the text (assuming again that the pattern fits into a word) and not on the actual text or the alphabet.

#### Approximate Matching

We now show how to adapt the previous algorithm to allow errors. (Baeza-Yates and Gonnet [2] showed how to handle only mismatches by essentially counting  $k$  of them with a  $\log_2 k$  size counter, but they did not handle insertions or deletions.) We start with a very simple case: only one insertion is allowed into the pattern at any position. In other words, we want to find all intervals of size at most  $m + 1$  in the text that contain the pattern as a subsequence. We define the  $R$  and  $S$  arrays as before, but now we have two possibilities for each prefix match. We can have an exact match or a match with one insertion. Therefore, we introduce another array, denoted by  $R_j^1$ , which indicates all possible matches up to  $t_j$  with at most one insertion. More precisely,  $R_j^1[i] = 1$  if the first  $i$  characters of the pattern match  $i$  of the last  $i + 1$  characters up to  $j$  in the text. If we can maintain both  $R$

and  $R^1$  then we can find all matches with at most one insertion:  $R_j[m] = 1$  indicates that there is an exact match and  $R_j^1[m] = 1$  indicates that there is a match with at most one insertion (sometimes both will equal 1 at the same time).

The transition for the  $R$  array is the same as before. We need only to specify the transition for  $R^1$ . There are two cases for a match with at most one insertion of the first  $i$  characters of  $P$  up to  $t_{j+1}$ :

I1. There is an exact match of the first  $i$  characters up to  $t_j$ . In this case, inserting  $t_{j+1}$  at the end of the exact match creates a match with one insertion.

I2. There is a match of the first  $i - 1$  characters up to  $t_j$  with one insertion and  $t_{j+1} = p_i$ . In this case, the insertion is somewhere inside the pattern and not at the end.

Case I1 can be handled by just copying the value of  $R$  to  $R^1$  and case I2 can be handled with a right shift of  $R^1$  and an AND operation with  $S_i$  such that  $s_i = t_{j+1}$ . So, to compute  $R_j^1$  we need one additional shift (the shift of  $R$  is done already), one AND operation and one OR operation. An example (with the same pattern and text as the example for the exact matching) is given in Table 2.

Consider now allowing one deletion from the pattern (and no insertions). We will define  $R$ ,  $R^1$  (which now indicates one deletion), and  $S$  as before. There are again two cases for a match with at most one deletion of the first  $i$  characters of  $P$  up to  $t_{j+1}$ :

D1. There is an exact match of the first  $i - 1$  characters up to  $t_{j+1}$  (which is indicated by the new value of the  $R$  array  $R_{j+1}[i - 1]$ ). This case corresponds to deleting  $p_i$  and matching the first  $i - 1$  characters.

D2. There is a match of the first  $i - 1$  characters up to  $t_j$  with one deletion and  $t_{j+1} = p_i$ . In this case, the deletion is somewhere inside the pattern and not at the end.

Case D2 is handled as before (it is exactly the same), and case D1 is handled by a right shift of the new value of  $R_{j+1}$ .

Finally we will consider a substitution. That is, we allow replacing one character of  $P$  with one character of  $T$ . (We can achieve substitution with one deletion and one insertion, but in many cases we want substitution to count as only one error.) We again have two cases:

S1. There is an exact match of the first  $i - 1$  characters up to  $t_j$ . This case corresponds to substituting  $t_{j+1}$  with  $p_i$  (whether or not they are

equal—the equality will be indicated in  $R$ ) and matching the first  $i - 1$  characters.

S2. There is a match of the first  $i - 1$  characters up to  $t_j$  with one substitution and  $t_{j+1} = p_i$ . In this case, the substitution is somewhere inside the pattern and not at the end.

Case S2 is again the same. Case S1 corresponds to looking at  $R_j[i - 1]$  as opposed to looking at  $R_{j+1}[i - 1]$  in case D1. Still very few operations cover one substitution as well.

We are now ready to consider the general case of up to  $k$  errors, where an error can be either an insertion, a deletion, or a substitution (the Levenshtein or the edit-distance measure). Overall, instead of one additional  $R^1$  array, we will maintain  $k$  additional arrays  $R^1, R^2, \dots, R^k$ , such that array  $R^d$  stores all possible matches with up to  $d$  errors. We need to determine the transition from array  $R_j^d$  to  $R_{j+1}^d$ . There are four possibilities for obtaining a match of the first  $i$  characters with  $\leq d$  errors up to  $t_{j+1}$ :

- There is a match of the first  $i - 1$  characters with  $\leq d$  errors up to  $t_j$  and  $t_{j+1} = p_i$ . This case corresponds to matching  $t_{j+1}$ .
- There is a match of the first  $i - 1$  characters with  $\leq d - 1$  errors up to  $t_j$ . This case corresponds to substituting  $t_{j+1}$ .
- There is a match of the first  $i - 1$  characters with  $\leq d - 1$  errors up to  $t_{j+1}$ . This case corresponds to deleting  $p_i$ .
- There is a match of the first  $i$  characters with  $\leq d - 1$  errors up to  $t_j$ . This case corresponds to inserting  $t_{j+1}$ .

Let's denote  $R$  as  $R^0$ , and assume that  $t_{j+1} = s_c$ . Overall, we have the

**Table 1.** An example of exact matching and the corresponding masks

	a	a	b	a	a	c	a	a	b	a	c	a	b	a	b	c
a	1	1	0	1	1	0	1	1	0	1	0	1	0	1	0	0
a	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0
b	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0
a	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0
c	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
	a.								b.							

**Table 2.** An example for approximate matching with one insertion

	a	a	b	a	a	c	a	a	b	a	c	a	b		a	a	b	a	a	c	a	a	b	a	c	a	b
a	1	1	0	1	1	0	1	1	0	1	0	1	0	a	1	1	1	1	1	1	1	1	1	1	1	1	1
a	0	1	0	0	1	0	0	1	0	0	0	0	0	a	0	1	1	1	1	1	1	1	1	0	0	0	0
b	0	0	1	0	0	0	0	0	1	0	0	0	0	b	0	0	1	1	0	0	0	0	1	1	0	0	0
a	0	0	0	1	0	0	0	0	0	1	0	0	0	a	0	0	0	1	1	0	0	0	1	1	0	0	0
c	0	0	0	0	0	0	0	0	0	0	1	0	0	c	0	0	0	0	0	1	0	0	0	0	1	1	0
	R													R <sup>1</sup>													

following expression for  $R_{j+1}^d$ :

$$\begin{aligned} R_0^d &= 11..100..000 \text{ } d \text{ ones.} \\ R_{j+1}^d &= Rshift[R_j^d] \text{ AND } S_c \text{ OR} \\ &\quad Rshift[R_{j+1}^{d-1}] \text{ OR} \\ &\quad Rshift[R_{j+1}^{d-1}] \text{ OR } R_j^{d-1} \\ &= Rshift[R_j^d] \text{ AND } S_c \text{ OR} \\ &\quad Rshift[R_j^{d-1}] \text{ OR } R_{j+1}^{d-1} \text{ OR} \\ &\quad R_j^{d-1}. \end{aligned} \quad (2.1)$$

Overall, we have a total of two shifts, one AND, and three ORs for each  $R^d$ .

The time and space analysis of the algorithm is as follows. Denote the word size by  $w$ . The preprocessing requires  $O(m|\Sigma|)$  time and space, which is the same as the exact matching algorithm, plus  $O\left(k\left\lceil\frac{m}{w}\right\rceil\right)$  to initialize the  $k$  vectors. During the matching process, the algorithm spends  $O\left(k\left\lceil\frac{m}{w}\right\rceil\right)$  steps per character. Therefore, the total running time is  $O\left(nk\left\lceil\frac{m}{w}\right\rceil\right)$ . In cases where  $m$  is not too large, which are the cases the algorithm was designed for, the running time is effectively  $O(nk)$ .

An important feature of this algorithm is that it can be relatively easily extended to several more complicated patterns. Several such extensions will be described.

#### An Improvement to the Main Algorithm

If the number of errors is small compared to the size of the pattern, then we can improve the running time sometimes by what we call *the partition approach*. Suppose again that the pattern  $P$  is of size  $m$  and that at most  $k$  errors are allowed.

Let  $r = \left\lfloor \frac{m}{k+1} \right\rfloor$ , and let  $P_1, P_2, \dots$ ,

$P_{k+1}$  be the first  $k+1$  blocks of  $P$  each of size  $r$ . In other words,  $P_1 = p_1 p_2 \dots p_r$  and  $P_j = p_{(j-1)r+1} \dots p_{jr}$ . If  $T$  contains a match of  $P$  with at most  $k$  errors, then at least one of the  $P_j$ 's must be matched exactly. We can search for all  $P_j$ 's at the same time (we discuss how to do that shortly) and, if one of them matches, then we check the whole pattern directly but only within a neighborhood of

size  $m$  from the position of the match. Since we are looking for an exact match, there is no need to maintain all  $k$  of the  $R^d$  vectors. This scheme will run fast if the number of exact matches to any one of the  $P_j$ 's is not too high. The number of such matches depends on many factors including the size of the alphabet, the actual text, and the values of  $r$  and  $m$ .

For example, if  $r = 1$ , then we will need to check any time there is a character match, which is probably too often. On the other hand, if  $r = 3$ ,  $m = 12$  (which implies  $k = 3$ ), the alphabet size is 26, and the text is uniformly random (i.e., each character appears with the same probability), the expected number of matches of any of the  $P_j$ 's is about 0.02% of the time. In this case, it is obviously advantageous to search for exact matches and use the approximate scheme only for the rare occasions at which a match occurs. The running time in this case is essentially the same as the running time of a search without errors. (Experiments using this partition scheme for different alphabet sizes are given later.)

The main advantage of this scheme is that the algorithm for exact matching presented earlier can be adapted in an elegant way to support it. We illustrate the idea with an example. Suppose the pattern is ABCDEFGHIJKL ( $m = 12$ ) and  $k = 3$ . We divide the pattern into  $k+1 = 4$  blocks: ABC, DEF, GHI, and JKL. We need to find whether any of them appears in the text. We create one combined pattern by interleaving the four blocks: ADGJB EHKCFIL. We then build the mask vector  $R$  as usual for this interleaved pattern (see section on Exact Matching). The only difference is that, instead of shifting by one in each step, we shift by four! There is a match if any of the last four bits is 1. (When we shift we need to fill the first four positions with 1s, or better yet, use shift-OR.) Thus, the match for all blocks can be done exactly the same way as

regular matches and it takes essentially the same running time.

#### Extensions

An important feature of our algorithm is its flexibility. In addition to asking about a single string, the algorithm supports range of characters (e.g., "0-9"), complements (e.g., everything except blank), arbitrary sets of characters (e.g., {a,e,i,o,u}), unlimited "wild cards," and combinations. Searching for several strings at the same time is also possible, although the size of the pattern becomes the sum of the sizes of the different strings (and might thus require more than one word to represent). Except for the unlimited wild cards, all these extensions were developed by Baeza-Yates and Gonnet [2] for exact matching. We can apply them all to approximate matching in a straightforward manner. This is a major strength of the algorithm: most extensions (at least all those that we tried) that work for exact matching can also be used for approximate matching. Previous algorithms for approximate matching with complements or wild cards, for example, were quite complicated (see, for example, [1, 18]). The running time for all these extensions is the same as the running time of the basic algorithm given earlier.

In addition to these extensions, we also support two extensions that are specific to approximate matching. One deals with patterns in which some parts need to match exactly and other parts may contain errors. Another extension is matching with some nonuniform costs; for example, we can define the cost of insertions to be twice that of deletions.

The most complicated patterns we can handle are arbitrary regular expressions. The ability to efficiently match regular expressions allowing errors enabled us to extend the grep family and implement the approximate pattern-matching tool *agrep*. We now describe the extensions in more detail.

## Sets of Characters

Replacing one character with a set of allowable characters is very easy to achieve with this algorithm (as was shown in [2]). Suppose the pattern we want to find is ABC followed by one digit followed by XYZ, and that we allow up to  $k$  errors. We denote this pattern by ABC[0–9]XYZ. The only thing we need to do to accept any digit in the fourth position is to put 1 in the fourth position in the  $S$  arrays for all digits. That is, in the preprocessing stage, when we decide for each character the positions that this character matches in the pattern, we include all the characters in the set within that position. The rest of the algorithm is identical with the regular algorithm. A complement of a character is a special case of a set of characters and it can obviously be handled in the same way.

## Wild Cards

A single wild card is a symbol that matches all characters. As such, it is a special case of a set of characters and can be handled as we discussed in the previous section. Sometimes, however, we want to indicate that we allow an unbounded number of characters to appear in the middle of the pattern (or even do it several times in the middle of the pattern). This case requires modifying the algorithm slightly. Let the pattern be  $P = p_1 p_2 \dots p_m$ , and assume that the positions of “#” (which indicates unlimited wild cards in agrep) are after the characters  $p_{i_1}, p_{i_2}, \dots, p_{i_r}$ . (There is no reason to have two #’s in a row.) Let  $S^\#$  be a bit array that has 1 in exactly the positions  $i_1, i_2, \dots, i_r$ . The effect of putting a “#” following  $p_i$  is as follows. If we are scanning  $t_j$  and we find a match with up to  $d$  errors that ends at  $p_i$ , then later when we scan  $t_r$ , for any  $r > j$ , we can start matching  $t_r$  to  $p_{i+1}$  no matter how many characters we skipped. In other words, if at some point there is a match up to  $p_i$  then this match is always valid later on (because all the characters later on can be considered as part of the “#”).

We can adjust the algorithm for this case as follows. At each step, we apply the regular algorithm to compute all the  $R$  arrays. That is, we compute  $R_j^0, R_j^1, \dots, R_j^d$  using (2.1). Then, for each  $i, 1 \leq i \leq d$ , we set  $R_j^i = R_j^i \text{ OR } [R_{j-1}^i \text{ AND } S^\#]$ . This step corresponds to the action “if at any point, there is a 1 entry in  $R^i \text{ AND } S^\#$ , then this entry should remain 1 from now on.”

## Unknown Number of Errors

In some cases, we do not know the number of errors *a priori*. We would like to find all occurrences of the pattern with the *minimal* number of errors possible. The algorithm can be extended to this case as follows. We first try to find the pattern with no errors. If we are unsuccessful, we try with one error, then with three errors, then with 7 errors, and so on, essentially doubling the number of errors (and adding one) at each attempt. If the number of errors turns out to be  $k$ , then the running time will be  $O(1 \cdot n + 2 \cdot n + 4 \cdot n + \dots + 2^b \cdot n)$ , where  $2^b$  is the first power of 2 greater than  $k$ . In the worst case, we perform 4 times as many operations as we would have had we known  $k$  (in most cases, the factor is actually 2 or 3).

## A Combination of Exact and Approximate Matching

Sometimes we do not want to allow parts of the pattern to have errors. For example, we may look for license plate ABC123, and we know that the letters are correct but the numbers may have one error in them. We denote this pattern by  $\langle \text{ABC} \rangle 123$ . We can modify the algorithm to shield parts of the pattern from having any errors in them. Let’s assume that  $I$  is the set of indexes in the pattern where no error is allowed, and let  $M$  be a masking array (of size  $m$ ) that has a 0 in the indexes of  $I$  and a 1 otherwise. We would like to modify (2.1) such that insertions, deletions, and substitutions can only occur outside of  $I$ . This is done by masking these cases with  $M$ . The expression in

(2.1) is changed to

$$\begin{aligned} R_{j+1}^d &= [R_{\text{shift}}[R_j^d] \text{ AND } S_c] \\ \text{OR } &[[R_{\text{shift}}[R_j^{d-1} \text{ OR } R_{j+1}^{d-1}] \\ \text{OR } &R_j^{d-1}] \text{ AND } M] \end{aligned} \quad (2.2)$$

## Nonuniform Costs

The edit distance measure assumes that insertions, deletions, and substitutions all have the same cost. But in some cases, we want to allow fewer deletions, say, than substitutions, or maybe no deletions at all. The algorithm can be extended, albeit in a limited way, to the case in which each operation has a different cost. We illustrate this extension with an example: Suppose that substitutions add 1 to the distance, but insertions and deletions add 3 each. Insertions and deletions are handled in cases 4 and 3 (see section 2). Insertions contribute the OR of  $R_j^{d-1}$  and deletions contribute the OR of  $R_{\text{shift}}[R_j^{d+1}]$  (2.1). We would like them to cost 3 times as much. In other words, a deletion or insertion that leads to a match with  $d$  errors should come from a match with  $d - 3$  errors. This can be achieved by simply replacing the  $d - 1$  in both expressions with  $d - 3$ . This modification is very simple and it does not add to the running time; however, it works only for small integer costs. (Another algorithm that allows different costs for different positions or different characters appears in [14].)

## A Set of Patterns

If we have several patterns and we want to find all occurrences of any of them, then we can either search them one at a time or together. The advantage of searching for all of them together is that it can be done in one scan (and in one command). Suppose we are looking for  $P_1, P_2, \dots, P_r$ . We concatenate all the patterns and put them in one array (using as many words as needed), and apply the algorithm on that array with the following modifications. Let  $M$  be a bit array the size of the combined pattern, and let bit  $i$

be 1 if and only if  $i$  corresponds to the first character of any of the patterns. For each  $S \in \Sigma$ , we build two bit arrays. The first,  $S_s$  is identical with the one we previously described. It is used to determine if a match occurs. The second array  $S'_s = S_s \text{ AND } M$ . It indicates whether  $s$  is the first character of any pattern. If so, then we must start the match at that pattern: we do not want to depend on the end of the previous pattern. Thus, after we compute  $R_j$ , we OR it with  $S'_s$  (where  $s = t_j$ ).

We compute the rest of the  $R$  arrays as before, except that in each step we OR them to a special mask that sets the first  $d$  bits in  $R^d$  of each separate pattern to 1; this allows  $d$  initial errors in each pattern. (This is not the most efficient way to solve this problem, but it is reasonably simple.) This case is a special case of patterns as regular expressions, which we will discuss shortly. (Version 2 of agrep includes a new algorithm for multipatterns, which uses Boyer-Moore-type filtering; it can handle very large patterns [24].)

### Long Patterns

Suppose that the pattern occupies several words and it is a simple string. The algorithm proceeds in the same fashion by computing the  $R^d$  arrays for all words. However, unless the number of errors is large, the first part of the pattern will not be matched quite often. If there is no match with  $k$  errors starting after position  $r$  of the pattern, then there is no need to maintain the  $R$  arrays corresponding to positions larger than  $r$  (their values will be 0). Thus, most of the time there will be no need to maintain the  $R^d$  arrays for the right side of the pattern. We only need to be alerted when the last bit of the last  $R^d$  array that we maintain gets the value of 1. In that case, we start maintaining the  $R_d$  arrays for the next part of the pattern. This improvement does not work for sets of patterns or regular expressions.

### Regular Expressions

The algorithm can be extended to

allow any regular expression as a pattern. We describe the method here only briefly; for more details see [23]. Other algorithms for approximate matching to regular expressions can be found in [17, 22, 25]. First, we illustrate the algorithm with a simple example. We do not try to optimize the algorithm here; we try to make it as simple as possible to describe. Let the pattern be  $P = ab(cd|e)^*fg$  (i.e., starting with  $ab$  and ending with  $fg$  with any number of either  $cd$  or  $e$  in between). This regular expression is translated to the nondeterministic finite automata shown in Figure 1 (for more on such translations, see [9]). We now assign a bit array to represent the automata. We number the states including the null states that do not correspond to any character. This "linearizes" the automata. Each state corresponds to one entry in the array. Thus, for  $P$  we have an array of size 11. Notice that all the non- $\epsilon$  moves go to the next state and thus can be handled by essentially a shift and an AND operation. We need to find a way to deal with arbitrary jumps required by the  $\epsilon$  moves (e.g., from state 2 to state 8) and with "nonjumps" that happen to be in consecutive states (e.g., from state 5 to state 6). The nonjumps can be handled easily with a mask. The arbitrary jumps are harder to handle. The meaning of an  $\epsilon$  move from state  $i$  to state  $j$  is that if, at any point, we match up to state  $i$  then the same match holds also up to state  $j$ . In other words, if there is a 1 corresponding to state  $i$  in the array, then the  $\epsilon$  move from  $i$  to  $j$  implies that there should be a 1 corresponding to state  $j$ . The main observation is that a given bit array and set of  $\epsilon$  moves completely determine the value of the bit array after the  $\epsilon$  moves are taken. Thus, the set of  $\epsilon$  moves defines a function that maps a bit array to another. We need to be able to implement this function efficiently.

Let  $f$  denote the function that maps one bit array to another by applying all the  $\epsilon$  moves. We divide the bit array into bytes (i.e., groups

of 8 bits each). Consider the first 8 bits of the bit array. The values of these bits determine which 1's should be set when we apply the  $\epsilon$  moves on states 1 to 8. Since there are only 256 ( $=2^8$ ) possible values for 8 bits, we can preprocess all possibilities and construct a table of size 256 which will hold, for each possible byte, the whole bit array with 1s only in places corresponding to the  $\epsilon$  moves. (We need the whole array and not just the first 8 bits, because there might be forward jumps.) We can do that for each byte. Given now a current value of  $R$ , we first apply the regular algorithm, taking care of regular non  $\epsilon$  moves, then we divide the array into bytes, find the corresponding arrays in the tables (we have one table per byte), and OR all of them to  $R$ . This implements all the jumps, and if the pattern occupies no more than 32 bits (as is often the case), only four more steps are required. (If the text is large, it is worthwhile to preprocess 16 bits for a table of size 65536 and half as many steps; this is the choice made in agrep.)

The running time of this algorithm is  $O\left(nk \left\lceil \frac{2p}{\log_2 n} \right\rceil \cdot \left\lceil \frac{p}{w} \right\rceil\right)$ , where  $p$  is the size of the regular expression. If the regular expression is not too large and  $n$  is large, the effective running time is  $O(nk)$ .

### Experimental Results

Here we present some experimental results involving the algorithms discussed in the previous sections. The purpose of these experiments is to show that our algorithms are very efficient for small patterns. We used only patterns that are smaller than the computer word size, thus the comparisons to other algorithms are not general and may not be fair. All tests were run on a SUN SparcStation II running Unix. The numbers given here should be taken with caution. Any such results depend on the architecture, the operating system, and the compilers used. Note also that the current version of agrep employs some newer algorithms, so its current

performance is sometimes better than the results in this section.

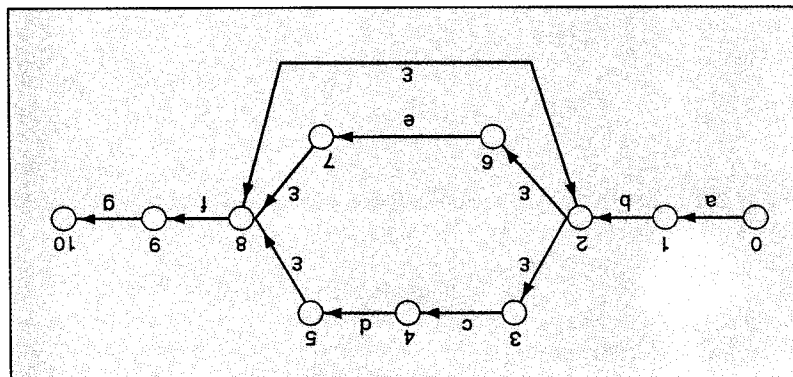
We tested the partition algorithm against two other algorithms, one by Ukkonen [20] (which we implemented) and one by Galil and Park [6] (labeled MN2; the program was provided for us by W.I. Chang) which is based on another technique by Ukkonen [21]. These two algorithms are both improvements of the basic dynamic programming algorithm and their expected running time is  $O(nk)$ . We used random text (of size 1 million) and pattern (of size 20), and two different alphabet sizes. In this case, since we use the idea of partitioning the pat-

tern, the size of the alphabet makes a big difference. A large alphabet leads to very few accidental exact matches, thus the running time is essentially the same as the one for exact matching. A small alphabet leads to many matches and the algorithm's performance degrades. The case of binary alphabet serves as the worst case for this purpose. Results are shown in Table 3.

The second test was for more complicated patterns, including some of the extensions discussed in the previous section. (Anything within the  $\langle \rangle$  brackets must match exactly; a  $\#$  stands for a wild card of arbitrary length; a  $;$  serves as

the Boolean AND operation, namely all patterns must appear within the same line; a  $|$  is the regular expression union operation; and a  $*$  is the Kleene closure.) The results are given in Table 4 (the file was a bibliographic text of size 1MB). Another algorithm for approximate matching to arbitrary regular expressions is by Myers and Miller [17]. (Recently, we improved some parts of this algorithm [25].) The running times of the Myers and Miller algorithm for the cases we tested, which include only small regular expressions and small errors, are more than an order of magnitude slower than our algorithm; however, this is not a fair comparison because their algorithm can handle more general cost functions and its running time is independent of the number of errors.

**Figure 1.** The nondeterministic automata corresponding to  $ab(cd|e)^*fg$



**Table 3.** Approximate string matching of simple strings

	Partition algorithm		MN2		Uk85a	
	$\Sigma = 2$	$\Sigma = 30$	$\Sigma = 2$	$\Sigma = 30$	$\Sigma = 2$	$\Sigma = 30$
$k = 0$	0.35	0.35	1.21	0.98	2.36	0.90
$k = 1$	0.52	0.38	3.03	2.42	5.01	2.06
$k = 2$	1.78	0.38	4.94	3.87	7.98	3.19
$k = 3$	2.56	0.39	6.68	5.33	11.80	4.38
$k = 4$	3.83	0.41	8.72	6.89	13.40	5.55
$k = 5$	4.42	0.42	10.41	8.28	15.45	6.77
$k = 6$	5.13	0.73	11.83	9.75	17.07	7.99

**Table 4.** Approximate matching of complicated patterns

pattern	$k = 0$	$k = 1$	$k = 2$	$k = 3$
Homogenous	0.35	0.39	0.41	0.47
<Hom>ogenous	0.53	1.10	1.42	1.74
JACM; 1981; Graph	0.53	1.10	1.43	1.75
Prob#tic; Algo#m	0.55	1.10	1.42	1.76
<[C]JACM>; Prob#tic; trees	0.54	1.11	1.43	1.75
(<[23]> -[23]*<B>).*<Tr>ees	0.66	1.53	2.19	2.83

## A Description of Agrep

Having such flexible algorithms motivated us to develop an approximate matching tool for Unix. Agrep has been in wide use since June 1991, and it has gone through two major revisions. It includes several new algorithms in addition to the ones described in this article (e.g., a new multipattern algorithm and an improved approximate string matching for simple strings). It is used similarly to grep/egrep/fgrep and it supports most of the features of the grep family (although it is not 100% compatible), but it supports many additional features (mostly but not all related to approximate matching). Agrep is at least as fast and in many cases faster than all other greps that we know of. One notable difference between agrep and the previous greps (besides the ability to search approximately) is that agrep is *record oriented* (rather than line oriented). A record (e.g., a paragraph, a mail message) is defined by the user (with the default being a line). Agrep outputs all records that match the query (see the  $-d$  option described below). Agrep is available, free of charge, by anonymous

ftp from cs.arizona.edu (IP number 192.12.69.5). Next, we describe the unusual features of agrep that are not found in similar programs. A complete description is given in the manual pages distributed with agrep (see also [24]).

-d 'delim'

delim is a user-defined symbol (or string) for record delimiter (the default is the new-line symbol). This enables searching paragraphs (in which case delim = '\$\$') or mail messages (delim = "From").

-k finds all occurrences with at most  $k$  errors (insertions, deletions, or substitutions), where  $k$  is a positive integer.

-B finds all matches with minimum number of errors (e.g., if the best match has 2 errors, this is the same as running agrep with -2, but of course 2 is unknown).

-Dc each deletion counts as  $c$  errors;  $c$  must be a nonnegative integer; the default value of  $c$  is 1

-Ic each insertion counts as  $c$  errors;  $c$  must be a nonnegative integer; the default value of  $c$  is 1

-Sc each substitution counts as  $c$  errors;  $c$  must be a nonnegative integer; the default value of  $c$  is 1

### Examples of using agrep

agrep -l -l2 -D2 555-1234 phone-list

finds all numbers that differ from 555-1234 in at most one digit; setting the cost of insertions and deletions to 2 prevents them in this case.

agrep -B -d '\$\$' Shmidth address-list

finds all addresses (assuming the addresses are separated by a blank line) containing the best match to Shmidth.

agrep -d "From " 'breakdown;arpanet' mail-file

outputs all mail messages (the pattern "From " separates mail messages in a mail file) that contain *breakdown* and *arpanet* (the symbol ; stands for the Boolean AND).

agrep -B -w otomaton/usr/dict/words

finds the closest words in the dictionary to *otomaton*; the -w option forces the match to be against the whole word rather than a part of it. (The best match has two errors and there are two of them: *otomaton* and *tomato*.)

agrep -d '\$\$' -l

'<word1> <word2>'

finds all paragraphs that contain word1 followed by word2 with at most one error in place of the blank between the words (the <> indicate that no error is allowed inside). In particular, if word1 is the last word in a line and word2 is the first word in the next line, then the space will be substituted by a new line symbol and it will match. Thus, this is a way to overcome separation by a new line. Note that -d '\$\$' (or another delim that spans more than one line) is necessary, because otherwise agrep searches only one line at a time.

agrep -5 'a#b#c#d#e#f#g#h#i#j' /usr/dict/words

finds all words that have at least 5 of the first 10 letters of the alphabet in order; the # sign signals unlimited wild cards, which means here that any insertion is for free (there is actually a special option -p for that purpose). Try it; the answer starts with *academia* and ends with *sacriligious*, which must mean something.


### Conclusions

Searching text in the presence of errors is commonly done by hand—one tries all possibilities. This is frustrating, slow, and with no guarantee of success. The new algorithm presented in this article for searching with errors can alleviate this problem and make searching in general more robust. It also makes searching more convenient by not having to spell everything precisely. The algorithm is very fast and general and it should find numerous applications.

There is one important area of searching with errors that we did not address—searching an indexed file. Throughout the article we as-

sumed that the files are not indexed (preprocessed) in any way, thus a sequential scan through them is necessary. We believe that the problem of finding good indexing schemes that allow approximate search is the unresolved problem in this area. We note, however, that with the speed of current computers, scanning large files (up to tens of MB) can be done reasonably fast. One can argue that since the size of our data increases as our speed of processing it increases, waiting for faster computers will not help and indexing will always be necessary. This is certainly true for some applications, but not for all. Some applications have a reasonable upper bound on their size and sequential search for those applications is (and will remain) realistic.

### Acknowledgments

We thank Ricardo Baeza-Yates, Gene Myers, and Chunghwa H. Rao for many helpful conversations about approximate string matching and for comments that improved the manuscript. We thank Ric Anderson, Cliff Hathaway, and Shu-Ing Tsuei for their help and comments that improved the implementation of agrep. We also thank William I. Chang for kindly providing programs for some of the experiments. 

### References

1. Abrahamson, K. Generalized string matching. *SIAM J. Comput.* 16, 6 (1987), 1039–1051.
2. Baeza-Yates, R.A. and Gonnet, G.H. A new approach to text searching. *Commun. ACM* 35, 10 (1992). Preliminary version appeared in *Proceedings of the 12th Annual ACM-SIGIR Conference on Information Retrieval*, Cambridge, Mass. (June 1989), pp. 168–175.
3. Boyer, R.S. and Moore, J.S. A fast string searching algorithm. *Commun. ACM* 20, 10 (Oct. 1977), 762–772.
4. Chang, W.I. and Lawler, E.L. Approximate string matching in sub-linear expected time. *FOCS 90*, pp. 116–124.
5. Galil, Z. and Giancarlo, R. Data structures and algorithms for ap-



- proximate string matching. *J. Complex.* 4 (1988), 33–72.
6. Galil, Z. and Park, K. An improved algorithm for approximate string matching. *SIAM J. Comput.* 19 (Dec. 1990), 989–999.
  7. Gonnet, G.H. and Baeza-Yates, R.A. *Handbook of Algorithms and Data Structures*. Second ed. Addison-Wesley, Reading, Mass., 1991.
  8. Hall, P.A.V. and Dowling, G.R. Approximate string matching. *Comput. Surv.* (Dec. 1980), 381–402.
  9. Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass. (1979).
  10. Knuth, D.E., Morris, J.H. and Pratt, V.R. Fast pattern matching in strings. *SIAM J. Comput.* 6 (June 1977), 323–350.
  11. Landau, G.M. and Vishkin, U. Fast string matching with  $k$  differences. *J. Comput. Syst. Sci.* 37 (1988), 63–78.
  12. Landau, G.M. and Vishkin, U. Fast parallel and serial approximate string matching. *J. Algor.* 10 (1989).
  13. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.* (Feb. 1966), 707–710.
  14. Manber, U. and Wu, S. Approximate string matching with arbitrary costs for text and hypertext. *IAPR Workshop on Structural and Syntactic Pattern Recognition*, (Bern, Switzerland Aug. 1992).
  15. Manber, U. and Wu, S. Approximate pattern matching. *BYTE*. To be published Nov. 1992.
  16. Myers, E.W. An  $O(ND)$  difference algorithm and its variations. *Algorithmica* 1 (1986), 251–266.
  17. Myers, E.W. and Miller, W. Approximate matching of regular expressions. *Bull. Math. Bio.* 51 (1989), 5–37.
  18. Pinter, R. Efficient string matching with don't-care patterns. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, Eds., Springer-Verlag, Berlin, 1985.
  19. Tarhio, J. and Ukkonen, E. Approximate Boyer-Moore string matching. Tech. Rep. #A-1990-3, Dept. of Computer Science, University of Helsinki (Mar. 1990).
  20. Ukkonen, E. Finding approximate patterns in strings. *J. Algor.* 6 (1985), 132–137.
  21. Ukkonen, E. Algorithms for approximate string matching. *Inf. Control* 64 (1985), 100–118.
  22. Wagner, R.A. and Seiferas, J.I. Correcting counter-automaton-recognizable languages. *SIAM J. Comput.* 7 (1978), 357–375.
  23. Wu, S. Approximate pattern matching and its applications. Ph.D. dissertation, Dept. of Comput. Sci., University of Arizona, June 1992.
  24. Wu, S. and Manber, U. Agrep—A fast approximate pattern-matching tool. *Usenix Winter 1992 Technical Conference* (San Francisco Jan. 1992), pp. 153–162.
  25. Wu, S., Manber, U. and Myers, E.W. A Sub-Quadratic Algorithm for Approximate Regular Expression Matching, submitted for publication (May 1992).

**CR Categories and Subject Descriptors:** F.2.2 [Theory of Computing]: Numerical Algorithms and Problems—*pattern matching*; H.3.3 [Information Systems]: Information Search and Retrieval—*search process*; I.5.4 [Computing Methodologies]: Pattern Recognition—*text processing*

**General Terms:** Algorithms

**Additional Key Words and Phrases:** Approximate string matching, information retrieval, pattern matching, software tools, string searching

#### About the Authors:

**SUN WU** is a member of the technical staff at Bell Laboratory in Murray Hill, N.J. His research interests include approximate pattern matching, text processing and distributed database systems. **Author's Present Address:** AT&T Bell Labs, 600 Mountain Ave., Murray Hill, N.J. 07974

**UDI MANBER** is a professor of computer science at the University of Arizona, where he has been since 1987. His research interests include pattern matching, design for algorithms and computer networks. **Author's Present Address:** Dept. of Computer Science, University of Arizona, Tucson, Az. 85721; email: udi@cs.arizona.edu

Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, and by an NSF grant CCR-9002351.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/92/1000-083 \$1.50

## CARE plants the most wonderful seeds on earth.

Seeds of self-sufficiency that help starving people become healthy, productive people. And we do it village by village by village. Please help us turn cries for help into the laughter of hope.

